Chapter 11: Lossless Compression

A major difference between multimedia data and most other data is its size. Images and audio files take much more space than text for example. Video data is currently the single largest network bandwidth and hard disk space consumer. One of the earliest topics in multimedia was therefore compression. In fact, multimedia's history is closely connected to different compression algorithms because they served as enabling technologies for many applications. Even today, multimedia signal processing would not be possible without compression methods. A Blue Ray disk can currently store 50 Gbytes, but a 90-minute movie in 1080p HDTV format takes about 800 Gbytes (without audio). So how does it fit on the disk?

This chapter discusses compression's underlying mathematical principles, from the basics to advanced techniques.

Run-Length Coding

Before discussing what compression is and how you can develop algorithms to represent different types of content with the least amount of space possible, let's start with a simple and intuitive example. In addition to introducing what compression is, this example demonstrates a practical method that computer scientists often use to compress data with large areas of the same values.

Suppose we have a black and white image encoding the character 0. The black pixels are encoded with 1 and the white pixels are encoded with 0. So, it looks like this:

```
0000000000000000
0000011111100000
0000100000110000
0000100011010000
0000100110010000
0000111000010000
0000011111100000
0000000000000000
```

As is, the bitmap representation would take 16 × 8 = 128 zeros and ones. Say we want to cut the number of zeros and ones. There are several ways to do it.

First, observe that there are many zeros and ones in a row. We could represent these characters only by the number of consecutive zeros and ones starting with the zeros. The result would be:

```
21 6 9 1 5 2 8 1 3 2 1 1 8 1 2 2 2 1 8 6 21
```

The highest number represented is 21. So, we could represent each of the numbers in the above row with 5 bits: 101010011001001... and so on. We need 21 numbers in this encoding, so we can represent the digit in 105 bits, for a total savings of 23 bits. Of course, we would have to represent a bitmap containing more than two colors—say, 16—slightly differently. Consider the following one-line example:

```
RRRRRGGGBBBBBRRRRGB
```

Using a variant of the above described concept of representation, we would get:

5R3G5B4R1G1B

This method—called *run-length encoding*, or RLE—is used in many image formats, such as Windows BMP and TIFF, and, in a slightly modified version, on CDROMs because it is especially effective in representing data with large areas of the same values. Several unanswered questions remain, however. For example:

- Is RLE the best way to compress the above example?
- Would applying RLE to the file again further compress the bitmap?
- Finally, what is the minimum amount of space needed to represent this image?

Answering these questions requires a more in-depth discussion of the topic. In fact, it leads to an entire theory in mathematics, the so-called information theory. The following sections introduce the most important concepts in information theory.

Information Content and Entropy

To determine how far we can compress a certain piece of data, we first need a measurement for information. The smallest amount of information is the bit. A bit is a symbol that can be 0 or 1. Every string in the world can be reduced to bits, so we could say that one measure for information is the minimum number of bits needed to represent a string. But how can we calculate this number?

Assume we have a 64-bit-long string consisting of 63 zeros and 1 one. The one can be at any place in the string. We denote this place with an i, so i is a number between 0 and 63. Next, assume we read the string from the left to the right, symbol by symbol, until we find the one. We can then ask: At each character, what is the probability P that the next character will be a one or a zero, given that we have not yet found the one? Table 1 shows the probability P for each bit, from 1 to 64.

i	1	2	3	4	....	32	33	...	62	63	64
P(b_i = 0)	63/64	62/63	61/62	60/61		32/33	31/32		2/3	1/2	0
P(b_i = 1)	1/64	1/63	1/62	1/61		1/33	1/32		1/3	1/2	1

Table 1: The probability that a bit (with the symbol 0 or 1) will appear in a particular place on string (*b* = the bit, *i* = the bit's place on a string, and P = probability).

As the table demonstrates, the probability P depends only on the index until the one is found. Of course, the probability after reading 63 zeros is 1 and the probability after reading the one is 0. In other words, the zeros after a one are completely predictable and carry no information. We can also reverse this argument: The higher an event's probability, the less information it carries. This means the *information content* is inversely proportional to the probability of a symbol's occurrence. What's left is to count how many characters we actually need to represent the content. In the binary system, this is the number of digits needed. We therefore define the information content $h(x)$ as:

$$h(x) = \log_2 \frac{1}{P(x)} \quad (1)$$

The choice of a logarithmic base corresponds to the choice of a unit for measuring information. If we use base 2, the resulting units are bits (as in this example). If we use another alphabet (for example, the decimal system), we must adjust the base accordingly.

So, let's measure the information content for each symbol for the given example by inserting the values from Table 1 into the formula in Equation 1. Table 2 shows the results.

i	1	2	3	4	....	32	33	...	62	63	64
$h(b_i = 0)$	0.0227	0.023	0.0235	0.0238		0.444	0.458		0.585	1	0
$h(b_i = 1)$	6	5.977	5.954	5.9307		5	5.0666		1.585	1	0

Table 2: The information content of each symbol calculated from the probabilities in Table 1.

The sum of the information content of bits 1 to *n* is 6 when the *n*-th bit is a one. For example, *h*(00010 . . . 0) = 0.0227 + 0.023 + 0.0235 + 5.9307 + 0 + . . . + 0 = 6

So, in general, the formula is:

$$\sum_{n=1}^{i} h(b_n) = \left(\sum_{m=66-i}^{64} \log_2 \frac{m}{m-1}\right) + \log_2 \frac{1}{1/(65-i)} = \log_2 \left(\left(\prod_{m=66-i}^{64} \frac{m}{m-1}\right) * (65-i)\right) = 6 \quad (2)$$

The summation of the information content for each symbol reveals the string's total information content. In other words: We need a minimum of six bits to represent the string as described. To do this, all we need to save is the index i (a number between 0 and 63), which can be represented by a six-digit binary number.

There is a caveat, however. To measure the string's information content, we use probabilities, which requires knowledge about the data structure. In other words, if we don't know how many ones appear in the string and the string's total length, we can't calculate the information content. Again, this is intuitive. For a string to contain any information, someone or something must process and interpret the string. That is, a string must be put into a context to make sense.

Information content gives us a method to measure the number of bits required to encode a certain message. In practice, however, the message might not be known in advance. Instead, you might have a language and arbitrary messages encoded in that language. Formally, we can define a language as a set of symbols or words. Each word or symbol has an associated probability. The probability can usually be determined by the frequency of a symbol or word appearing in the language. In English, for example, the word "the" appears more often than the word "serendipity." As a result, the probability of an article ("the") appearing in a message is much higher than that of the word derived from the old Persian name for Sri Lanka ("serendipity's" origin).

So, what is the expected length of a message given a set of symbols and their probabilities? This measure—*entropy*—is defined as follows:

$$H(X) = \sum_i P(x_i) * h(x_i) \quad (3)$$

Entropy is the expected information content—not coincidentally similar to probability theory's expectation value. Entropy is an important measure used across scientific disciplines. Shannon's source coding theorem (Shannon 1948) gives the information theoretic background for entropy. The theorem shows that entropy defines a lower bound for the number of bits that can be used to encode a message in that alphabet without losing information. Entropy's value is maximized when the alphabet is uniformly distributed—that is, each character has the same probability. This is also often referred as chaos: Given a subset of a string, there is no possibility of a prediction about the next symbols because all symbols have equal probability. Random noise has this property and is therefore incompressible.

Entropy is a fundamental measure with analogies in thermodynamics, quantum mechanics, and other fields. In general, it can be described as a measure of chaos: The more chaotic a system, the higher its entropy. In information theory, this means that the more chaotic a string is, the more bits we need to encode it.

Compression Algorithms

Information content and entropy lets us measure a message's expected minimum length. However, given a string, how do you construct a code for that string that uses the minimal number of bits? Unfortunately, no single answer to this problem exists. There are many ways to construct the string and none of them achieves perfect results in all cases. Therefore, the best compression method depends on the data you are processing. The following sections present the most important compression algorithms.

Huffman Coding

Before we dig into the details of how to create good compression algorithms, let's quickly review the features we would expect from a good encoding:

•The code must be unambiguously decodable—that is, one coded message corresponds to exactly one decoded message.
•The code should be easily decodable—that is, you should be able to find symbol endings and the end of the message easily. Ideally, you should be able to decode it online (that is, as the symbols come in) without having to know the entire coded message.
•The code should be compact, only delimited by entropy.

In 1952, David A. Huffman invented a coding algorithm that produces a code fulfilling these requirements. *Huffman coding* is in frequent practical use and part of many standard formats. The algorithm's design is elegant, easy to implement, and efficient. Also, because of the vast mathematical work that's been done on tree structures, the algorithm is well understood.

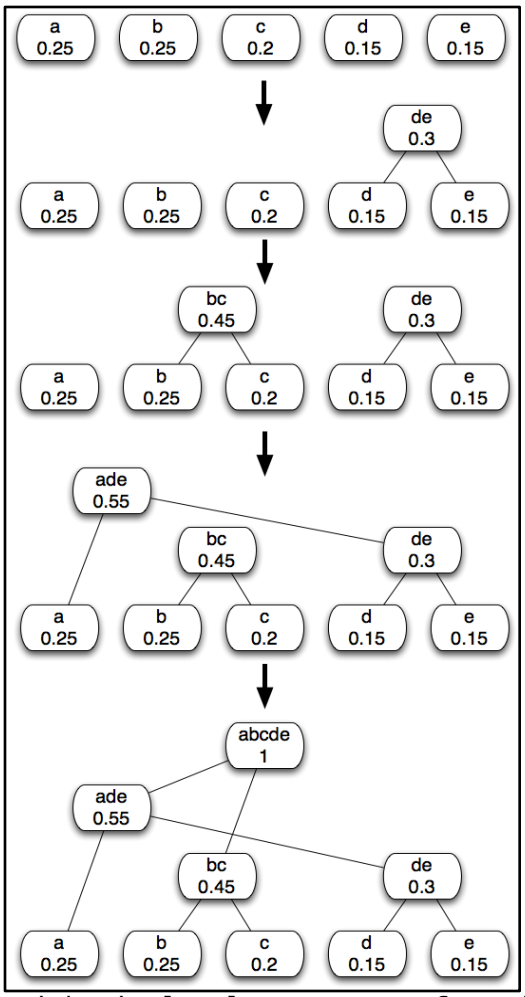
The algorithm constructs a binary tree in which the symbols are the leaves. The path from the root to a leave reveals the code for the symbol. When turning right on a node, a 1 is added to the code; when turning left, a 0 is added. The idea is to have long paths for symbols that occur infrequently and short paths for symbols that occur frequently.

Figure 1 illustrates the construction of a Huffman tree. In the beginning, the leave nodes containing the frequencies of the symbol they represent are unconnected. The algorithm creates a new node whose children are the two nodes with the smallest probabilities, such that the new node's probability equals the sum of the children's probability. The new node is treated as a regular symbol node. The procedure is repeated until only one node remains—the root of the Huffman tree.

Figure 1: Construction of a Huffman tree.

The simplest construction algorithm uses a priority queue in which the node with the lowest probability receives highest priority. The following pseudo code illustrates the process:

```
// Input: A list W of n frequencies for the symbols
// Output: A binary tree T with weights taken from W
Huffman(W, n):
```




```
Create list F with single-element trees formed from elements of W
WHILE (F has more than one element)
      Find T1, T2 in F with minimum root values
      Create new tree T by creating a new node with root value T1+T2
```

```
            Make T1 and T2 children of T
            Add T to F
Huffman := tree stored in F
```

You can now use the tree to encode and decode any message containing symbols of the leave nodes, as Figure 2 shows.

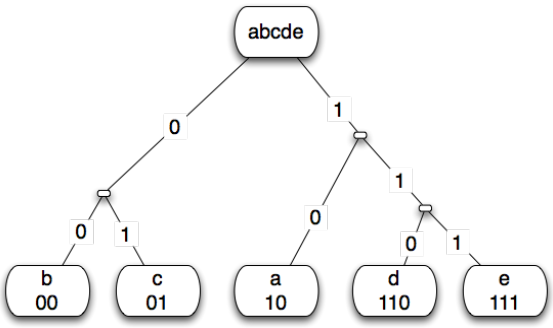


Figure 2: Codes generated by the Huffman tree

To encode a message, the algorithm traverses the tree completely and saves the paths into a lookup table. So, in our example, the encoding for the word “cab” is 011000. The following pseudo code describes the procedure:

```
// Input: A Huffman tree F and a string S containing only symbols in F
  // Output: A bit-sequence B
Encode_Huffman(F, S)
    Traverse F completely and create a hash table H containing tuples (c,p)
    // c = character, p = path from root to of F to character
    B := []
    FOREACH (symbol s in S)
          Add path H(s) to bit sequence B.
Encode_Huffman := B
```

To decode a message, the algorithm interprets the code as encoding a path from the root to a leaf node, where 0 means going right, and 1 means going left (or vice versa), as described in the following pseudo-code sequence:

```
// Input: A Huffman tree F and a bit sequence B
  // Output: An uncompressed string S
Decode_Huffman(F, B)
    Traverse F completely and create a hash table H containing tuples (p,c)
    // c = character, p = path from root to of F to character
    S := ""
    node n = root node of F
    FOREACH (bit b in B)
          if (b==0) n := left children of n
          if (b==1) n := right children of n
          if (n has no children)
                Add character represented by n to S
                n := root node of F
Decode_Huffman := S
```

Static Huffman compressors use a fixed tree for all incoming data; dynamic Huffman compressors serialize the tree, so that each file can be decoded using a different tree. The code generated by this algorithm is optimal in the sense that each symbol has the shortest possible representation for the given probability. However, the Huffman algorithm assumes that symbol frequencies are independent of each other. In reality, a "q," for example, is usually followed by a "u." In addition, the Huffman algorithm cannot create codes with a fraction of bits per symbol.

Lempel-Ziv Algorithms

The LZ family of algorithms (derived from their creators'—Abraham Lempel and Jacob Ziv—last names) is divided into two groups: successors of the LZ77 and successors of the LZ78 algorithm. Lempel and Ziv developed the base algorithms in 1977 and 1978, respectively. The algorithms use completely different approaches, despite their similar names. Most variations of the LZ algorithms can be identified by the third letter—for example, LZH or LZW. In contrast to Huffman coding, which is based on symbol probabilities, the algorithms account for repeated symbol combinations (aka words) in a document. The algorithm is best suited for large archives containing text and/or source code and two very popular image formats, GIF and PNG, use one variant of each of the algorithm. The LZ compression algorithms are therefore the single most often used compression algorithms of all times. Once one has understood both LZ77 and LZ78, the derivate algorithms are conceptually not very different. This section will therefore describe the two fundamental algorithms: LZ77 and LZ78.

LZ77 achieves compression by replacing portions of the data with references to matching data that have already passed through the encoder and decoder. This algorithm works with a fixed number of symbols—the *look-ahead buffer* (LAB)—that are to be coded. Additionally, the algorithm looks at a fixed number of symbols from the past—the *search buffer* (SB). To encode the symbols in the LAB, the algorithm searches the SB backward for the best match. The encoding (often called a *token*) is encoded by a tuple (L,D) defining the length and the distance from the current token to a past token. A token basically says, "each of the next L symbols is equal to the D symbols behind it in the uncompressed stream." In actual implementations, the distance D is often referred to as *offset*. The current chunk of processed symbols—that is, SB+LA—is often called a *sliding window*. Typical sizes for the sliding window are several kilobytes (2 kB, 4 kB, 32 kB, and so on).

Consider the example in Table 3. It encodes the string "cabbdcbaacbdddabda" using LZ77. The currently processed symbol is underlined.

		Sliding window			
Position	**Code to be processed**	**Look-ahead buffer (next three symbols)**	**Search buffer (previous seven symbols)**	**Already encoded**	**Token**
	cabbdcbaacbdddabda				
1	bdcbaacbdddabda	cab			(0,0,c)

2	dcbaacbdddabda	abb	c		(0,0,a)
3	cbaacbdddabda	bbd	ca		(0,0,b)
4	baacbdddabda	bdc	cab		(1,1,d)
6	acbdddabda	cba	cabbd		(5,1,b)
8	bdddabda	aac	cabbdcb		(6,1,a)
10	ddabda	cbd	bbdcbaa	ca	(4,2,d)
13	bda	dda	cbaacbd	cabbd	(1,2,a)
16		bda	acbddda	cabbdcba	(5,2,a)
19			dddabda	cabbdcbaacb	

Table 3: Encoding of the string “cabbdcbaacbdddabda” using the LZ77 compression algorithm.

The LAB is also used as the SB, as position 13 demonstrates. The original LZ77 algorithm uses a fixed-size SB so can use constant-bit-length tokens. The encoding is the final token. For our example, the encoding is 0,0c 0,0a 0,0b 1,1d 5,1b 6,1a 4,2d, 1,2a 5,2a

The following is the pseudo-code for an LZ-77 encoder:

```
// Input: A lookaheadbuffer LAB a substring of a message...
  // Output: An LZ77 encoding C
Encode_LZ77(LAB)
    C := ””
    WHILE (LAB not empty)
        p := position of the longest match in the window for the LAB
        l := length of the longest match
        Add the tuple (p, l) to C
        Add the first character in LAB to C
        Shift LAB by l
Encode_LZ77 := C
```

To decompress the code, the algorithm reads the constant-sized tokens. The distance and length always refer to already decoded values. Choose a string and try this algorithm for yourself **.** You may use the following pseudo code as a guidance:

```
// Input: A string C containing LZ77 code
  // Output: A string S containing the decoded code
Decode_LZ77(C)
    S := ””
    FOREACH (token in S)
        Read token and obtain the triple (p,l,c)
        // p = position, l=length, c=character
        Add to C the l characters from position length(C)-p
```

```
        Add to C the character c
Decode_LZ77 := S
```

As explained earlier, many variations of the LZ77 algorithm exist. LZR, for example, has an unrestricted search buffer and therefore variable-bit-length tokens. In LZH, a commonly used variant, the token values are compressed using a Huffman encoding. The currently most popular LZ77-based compression method is called DEFLATE, which is for example part of the very common Unix compression program "gzip".

DEFLATE combines LZ77 with Huffman coding, placing literals, lengths, and a symbol to indicate the end of the current block of data together in one alphabet. It places distances into a separate alphabet. The image format PNG (Portable Network Graphics, see also Chapter XXX), also uses a variation of DEFLATE. PNG combines DEFLATE with a filter to predict each pixel's value based on previous pixels' colors, subtracts the prediction from the actual value. Both encoder and decoder use the same prediction table. This way, only the prediction differences are transmitted. An image line filtered this way is usually more compressible than the raw image line because DEFLATE does not understand that an image is a 2D entity. It sees the image data as a stream of bytes, whereas the prediction accounts for neighboring pixels in all dimensions. We discuss this algorithm in more detail in Chapter XXX.

Whereas the LZ77 algorithm works on past data, the LZ78 algorithm attempts to work on future data. It achieves this by maintaining a dictionary. The input buffer forward-scans the input buffer and matches it against the dictionary. The algorithm scans the input buffer for the longest match of the buffer with a dictionary entry until it can no longer find a match. At this point, it outputs the location of the word in the dictionary (if one is available), the match length, and the character that caused a match failure. It then adds the resulting word to the initially empty dictionary. Table 4 shows an example in which the word "abacbabaccbabbaca" is compressed using LZ78.

Step	Input	Token	New dictionary entry/Index
1	a	0,a	a,1
2	b	0,b	b,2
3	ac	1,c	ac,3
4	ba	2,a	ba,4
5	bac	4,c	bac,5
6	c	0,c	c,6
7	bab	4,b	bab,7
8	baca	5,a	Baca,8

Table 4: Compressing the string "abacbabaccbabbaca" using the LZ78 algorithm.

Like in LZ77, the tokens are the actual coding. Therefore, the code for our example is "0a0b1c2a4c0c4b5a."

```
// Input: A string S
// Output: A string C containing the LZ78 code
Encode_LZ78(S)
    Start with empty dictionary D
    C := ""
    prefix := "" // stores the prefixes
    FOREACH (character c in S)
        IF (prefix+c is in D)
            prefix := prefix + c
        IF (prefix + c is not in D)
            Add the string prefix+c to D
            Add the index of prefix in D to C
            Add c to C
            prefix := c
Encode_LZ78 := C
```

Decompression of LZ78 tokens is similar to compression. The algorithm extends the dictionary by one entry when it decodes a token using the dictionary index and the explicitly saved symbol.

```
// Input: An LZ78 encoded string C
// Output: A string S containing the original message
Decode_LZ78(C)
    Start with empty dictionary D
    oldindex := indexvalue of the first token in C
    S := ""
    FOREACH (token t in C)
        index := indexvalue of t
        IF (D has entry with index)
            Add the string at index to S
            c := first character of the string at index
            Concatenate entry at oldindex and c and add to D
        IF (D has no entry with index)
            c := first character of the string at oldindex
            Concatenate entry at oldindex and c and add to D
            Concatenate entry oldindex with c and add to S
        oldindex := index
Decode_LZ78 := S
```

Both compression and decompression benefit from an easy-to-manage dictionary. Usually, you would use a tree in which each node has a certain number of children that equals the number of valid input symbols. In the original LZ78 algorithm, the dictionary's size is unrestricted. Therefore, the index values must be saved with a variable number of bits. The dictionary index length is rarely explicitly defined; the algorithm uses dictionary's size to determine the index's bit length. In other words, the algorithm allocates enough bits so that the largest index can be stored. For example, for a 24-bit dictionary, all we need is $[\log_2 24] = 5$ bits per index.

LZ78 has many variants too: LZC uses a maximum size for the dictionary. If it reaches the maximum number of entries, the algorithm continues with the current dictionary under the hope that the output file does not become too long. If it determines that the output length has passed a threshold, it recompresses the data by creating an additional dictionary. The Unix compress tool uses LZC. However, the LZC is patented so users must pay a license fee. LZW, a common LZ78

variant, does not store the following symbol explicitly but as the first symbol of the following token. The dictionary starts with all possible input symbols as first entries. This leads to a more compact code and lets users define the input symbols (which can vary in bit length). The popular Graphics Interchange Format (GIF) uses the LZW variant. Although initially popular, enthusiasm for LZ78 dampened, mostly because parts of it were patent-protected in the United States. The patent for the LZW algorithm was strictly enforced and led to the creation of the earlier-mentioned patent-free PNG image format.

As mentioned earlier, the RLE algorithm is most useful when the same characters repeat often and Huffman compression is most useful when you can build a non-uniformly distributed probability model of the underlying data. The LZ algorithms are especially useful with text-like data—that is, data where strings of limited, but variable lengths repeat themselves. Typical compression ratios are (original:compressed) 2:1 to 5:1 or more for text files. In contrast to RLE and Huffman, LZ-algorithms need a certain input file size to amortize. Compressing a file with just a few bits, such as our example from the beginning of the chapter, won't yield a very good compression result. The Unix program "tar", for example, therefore concatenates all files into one large archive and then invokes "gzip" on the entire archive.

Arithmetic Coding

Arithmetic encoding approaches seek to overcome Huffman encoding's limitations—namely, that messages can only be encoded using an integer number of bits per symbol. JPEG image compression and other standards use arithmetic coding.

Arithmetic coding maps every symbol to a real number in the open interval between 0 and 1, formally $[0,1) \subset \mathrm{R}$. Because this interval contains nondenumerable infinite elements, every message can be mapped to one number in this interval. A special termination symbol denotes the end of a message. Without this symbol, you would not know when to stop the decoding process. To encode a message, arithmetic coding approaches partition the start interval [0,1) into subintervals sized proportionally to the individual symbols' probabilities. The algorithm is as follows.

Choose the symbol with the highest probability and split the interval according to its probability. Using the reminder of the interval, repeat the process until you reach the symbol with the lowest probability. Table 5 illustrates the process for the string "bac#" ("#" denoting the end symbol) with the probabilities given in the first row. The table also shows how the actual binary code is created by converting the decimal representation to binary.

x_i	A	B	c	#
$P(x_i)$	0.5	0.2	0.2	0.1
Step 1: read "b"				
Partition (decimal)	$[0,0.5)$	$[0.5,0.7)$	$[0.7,0.9)$	$[0.9,1)$
Partition (binary)	$[0,0.1)_2$	$[0.1,0.1\overline{0110})_2$	$[0.1\overline{0110},0.1\overline{1100})_2$	$[0.1\overline{1100},1)_2$
Step 2: read "a"				

Partition (decimal)	$[0.5,0.6)$	$[0.6,0.64)$	$[0.64,0.68)$	$[0.68,0.7)$
Partition (binary)	$[0.1,0.1\overline{1001})_2$	$[0.\overline{1001},0.1010001...)_2$	$[0.1010001...,0,1010111...)_2$	$[0.1010111...,0,\overline{10110})_2$
Step 3: read “c”				
Partition (decimal)	$[0.5,0.55)$	$[0.55,0.57)$	$[0.57,0.59)$ $[0.100100011...,$ $0.100101110...\ \frac{1}{1})_2$	$[0.59,0.6)$
Step 4: read #				
Partition (decimal)	$[0.57,0.58)$	$[0.58,0.584)$	$[0.584,0.588)$	$[0.588,0.59)$ = $[0.100101101...,$ $0.100101110...\ \frac{1}{1})_2$

Table 5: Steps involved in encoding the string “bac#” using arithmetic coding.

The message “bac#” is encoded as a number in the interval [0.588,0.59) or [0.100101101 . . . , 0.100101110 . . .) binary. More exactly formulated, this interval contains all messages starting with “bac.” The decode, however, stops because it reads the terminal symbol “#.” Because the start interval is open and does not contain the one, there is no need to transmit the numbers before the point. So the code for “bac#” is 10010111. The following pseudo code snippet illustrates the idea:

```
// Input: A string S containing a message, ending with a stop symbol
// Output: An arithmetically encoded string C,
  // a table P mapping probability ranges to symbols
Encode_Arith(S)
     FOREACH (character c in S)
         Get the frequency and assume as probability p[c]
     Create a table P that assigns characters to probability
       ranges in [0,1), each range with a size proportional to p[c]
     lower_bound := 0
     upper_bound := 1
     FOREACH (character c in S)
         current_range := upper_bound-lower_bound
         upper_bound := lower_bound+(current_range*upper_bound[c])
         lower_bound := lower_bound+(current_range*lower bound[c])
     C=upper_bound+lower_bound/2.0
Encode_Arith := (C, P)
```

To decode the message, the algorithm needs the input alphabet and the intermediate partitions. Given an encoded message, the algorithm then chooses the subinterval containing the code in each intermediate partition until the algorithm finds the termination symbol. The table with intermediate partitions is rarely transmitted; instead, it’s typically generated identically by the coder and decoder. The following lines of pseudo-code illustrate the decoding:

```
// Input: An arithmetically encoded string C,
  // a table P mapping probability ranges to symbols
```

```
// Output: A string S containing the orginal message
Decode_Arith(C, P)
    encoded_value := C
    WHILE (we have not seen the terminal symbol)
            s := symbol in P where encoded_value is within its range
            //remove effects of s from encoded_value
            current_range = upper_bound of c - lower bound of c
            encoded_value = (encoded_value-lower_bound of c)/current_range
            Add s to S
Decode_Arith := S
```

A major problem in implementing arithmetic codes is that the interval boundaries must be accurately represented. Standard processors use single- (32 bit) or double-precision (64 bit) floating-point numbers, but this representation is, of course, not precise enough. Different arithmetic encoders use different tricks to overcome this problem. Rather than try to simulate arbitrary precision (which is a possibility but very slow), most arithmetic coders operate at a fixed limit of precision. The coders round the calculated fractions to their nearest equivalents at that precision. A *renormalization* process keeps the finite precision from limiting the total number of symbols that can be encoded. Whenever the range decreases to the point at which the interval's start and end values share certain beginning digits, the coder sends those digits to the output, thus saving the digits in the CPU, where the interval boundaries shift left by the number of saved digits. This lets the algorithm add an infinite number of new digits on the right, even when the CPU can handle only a fixed number of digits.

Various multimedia data formats (such as JPEG) use variants of arithmetic coding. However, US patents cover several specific arithmetic coding techniques. For that reason, encoders and decoders of the JPEG file format typically only support Huffman encoding. In addition, although every arithmetic encoding implementation achieves a different compression ratio, most compression ratios vary insignificantly (typically within 1 percent). However, the CPU time varies greatly—easily an order of magnitude depending on the input. This runtime unpredictability is, of course, a major usability concern and therefore another reason for not choosing arithmetic coding.

Weakness of Entropy-based Compression Methods for Multimedia Data

All of the compression techniques we've described so far try to reconstruct the data in full, without losing any information, so are called *lossless* compression methods. Another name for these techniques, because they can be described by information theory, is *entropy encoders*. When entropy compression routines were developed, most of the data in computing systems were programs or text data. This does not mean that these compression methods cannot be used for images, sounds, or videos. However, the compression obtained when using LZ77 or other variants is usually a factor two or less. There are many reasons for the lower compression rates, including:

• Entropy is usually higher for multimedia signals than for text data and differs across files, even if the multimedia data contains no noise (think of the alphabet needed to store an English text versus the alphabet needed to store an amplitude-modified clean sinus signal).
• Because sampled signals contain noise, it is almost impossible to find repeating patterns, such as is done in LZx algorithms.

• Multimedia data is usually multidimensional. To leverage redundancies between neighboring pixels or frames requires knowing the data's basic structure. For example, you must know the image's resolution to know what the neighboring pixels are.
• Many algorithms—for example, arithmetic coding—could work well with some multimedia content; however, their asymptotic runtime behavior makes using them for multimedia practically prohibitive.

The next chapter looks at methods for overcoming these challenges.

Exercises

1.Give one example of content for which run-length encoding (RLE) would work very well and one for which it would work pretty badly.

2.Specify three implementations of RLE that handle short run lengths differently. Discuss their advantages and disadvantages.

3.What is the information content of a coin toss?

4.Show that it is impossible to construct a compression algorithm that takes an arbitrary input string and always produces a shorter output string without losing any information.

5.Show that applying one compression method repeatedly does not yield significantly better results than applying it only once.

6.Create a tool that measures a file's entropy. Use the tool to calculate the entropy of three different text files containing English text, source code, a binary program, an uncompressed image file (for example, TIFF), and an uncompressed audio file.

7.Many file archival utilities provided in today's operating systems first concatenate a set of files and then apply compression techniques on the entire archive rather than on each file individually. Why is this method typically advantageous?

8.Discuss the efficiency of Morse code.

9.Construct the Huffman tree for the word "Mississippi."

10.Write a dynamic-tree Huffman encoder/decoder in a programming language of your choice. Don't forget to encode the tree.

11.What is the minimum number of bits needed to represent an arbitrary Huffman tree?

12.Would it make sense to combine Huffman encoding with RLE encoding? If yes, give an example where this would be useful.

13.Compress the word "Mississippi" using LZ77 and LZ78 as described in this chapter.

14.Discuss the usefulness of using an LZx algorithm to compress a video file (images only).

15.How would you compress an audio file using LZ77? Define a filter that would allow for better compression.

16.Compress the word "Mississippi" using arithmetic coding.

17.Give an example of a symbol distribution in which arithmetic coding could result in a shorter output string than Huffman coding

18.Implement a simple arithmetic coder using the Unix tool "bc."

19.Explain how arithmetic coding accounts for repeating words.

20.Which of the methods—RLE, Huffman, arithmetic coding, or LZW—would you use for the following files: a text file, an image file containing a screenshot, a photograph containing a portrait, a midi file, a sampled audio file containing rock music, a sampled audio file containing a generated sinus waveform, a movie (images only), and a cartoon animation (images only).

Literature

-David MacKay, *Information Theory, Inference and Learning Algorithms*, Cambridge University Press, 2006.
-Khalid Sayood, *Introduction to Data Compression*, Morgan Kaufmann, 2nd edition, 2000.
-David Salomon, *Data Compression—The Complete Reference*, Springer, 2nd edition, 2000.

Web Links

•Compression Frequently Asked Questions: http://www.faqs.org/faqs/compression-faq/

Research Papers

-E. Shannon, “A Mathematical Theory of Communication,” *Bell System Technical Journal*, Vol. 27, July, October, 1948, pp. 379–423, 623–656.
-D.A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proceedings of IRE*, 1952, vol. 40, no. 10, pp. 1098-1101.
-J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337-343.
-J. Ziv and A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding,” *IEEE Transactions on Information Theory*, 1978, vol. 24, no. 5, pp. 530-536.
-T.A. Welsh, “A Technique for High-Performance Data Compression,” *Computer*, 1984, vol. 17, no. 6, pp. 8-19.
-I.H. Witten, R.M. Neal, and J.G. Cleary, “Arithmetic Coding for Data Compression,” *Communications of the ACM*, 1987, no. 30, vol. 6, pp. 520-540.